A bit of Standard Programming-Language Vocabulary

You've heard most of these terms before, but we'll try to define them a bit more thoroughly.

Three useful categories

Learning a programming language involves:

Syntax: The grammar rules defining a program (or fragment).

Semantics: The meaning of various programming fragments.

Pragmatics: How to *effectively* use language features, libs, IDEs, …

All three of these are important in how easy it is to easily write high-quality software.

For all categories, consider: Principle of least surprise.

Some vocabulary

Do *not* confuse the following four!

- *value*:
- *variable*:
- *type*:
- *expression*:

Some vocabulary

Do *not* confuse the following four!

- *value*: a datum – the fundamental piece of information that can be represented in the program

  E.g. `37` or `"hi"`. Values can be passed to functions, returned, stored in variables.
- *variable*:
- *type*:
- *expression*:

Some vocabulary

Do *not* confuse the following four!

- *value*: a datum – the fundamental piece of information that can be represented in the program

  E.g. `37` or `"hi"`. Values can be passed to functions, returned, stored in variables.
- *variable*: an identifer which, at run time, evaluates to some particular value.
- *type*:
- *expression*:

Some vocabulary

Do *not* confuse the following four!

- *value*: a datum – the fundamental piece of information that can be represented in the program

  E.g. `37` or `"hi"`. Values can be passed to functions, returned, stored in variables.
- *variable*: an identifer which, at run time, evaluates to some particular value.
- *type*: a set of values

  E.g. Java's `short` = {-32768,..., -1,0,+1,+2, ..., +32767}.
- *expression:*

Some vocabulary

Do *not* confuse the following four!

- *value*: a datum – the fundamental piece of information that can be represented in the program

  E.g. `37` or `"hi"`. Values can be passed to functions, returned, stored in variables.
- *variable*: an identifer which, at run time, evaluates to some particular value.
- *type*: a set of values

  E.g. Java's `short` = {-32768,..., -1,0,+1,+2, ..., +32767}.
- *expression*: a piece of syntax which evaluates to some particular value.

  E.g. `3+4*5` or `sqrt(16)`.

Some vocabulary (cont.)

- *parameter*: in a function-declaration: A local-variable, which is initialized when the function is called.
- *argument*: The value used to initialize a parameter, when calling a function.

```
(define (foo n) …) ; `n` is param.
(foo (+ 2 3)) ; 5 is an arg.
```

Some people use the terms interchangably; others use "formal parameter" and "actual parameter". But they're such useful, distinct concepts that I like having two terms for them.

Some vocabulary (cont.)

- *literal*: a value which literally appears in the source-code. E.g. Java `37` or `045` are both literals representing the value 37, which is of *type* `int`. And `37.`, `37d`, `37e0` are each literal `double` s. (But `pi` is not, nor `n+m`.)

  A compiler can generate the bit-representation of the literal value at compile-time.

  (We *will* often conflate a literal with the value it represents, and only say "literal" when we're emphasizing that we're dealing with syntax.)

trivia: Interning Java string-literals

Literals occur in the source-code text, and can be processed at compile-time. In Java, string literals are “interned”: If the same string-literal occurs twice, the the compiler is smart enough to only make one object(*), and use the same reference in both places.

```
"Cathay".substring(3).equals("hay")
```

Morever: string-literals with + are computed at compile-time.

(*) This optimization is only safe because Java strings are immutable.

trivia: Interning Java string-literals

Literals occur in the source-code text, and can be processed at compile-time. In Java, string literals are “interned”: If the same string-literal occurs twice, the the compiler is smart enough to only make one object(*), and use the same reference in both places.

```
"Cathay".substring(3).equals("hay") // true
```

Morever: string-literals with + are computed at compile-time.

(*) This optimization is only safe because Java strings are immutable.

trivia: Interning Java string-literals

Literals occur in the source-code text, and can be processed at compile-time. In Java, string literals are "interned": If the same string-literal occurs twice, the the compiler is smart enough to only make one object(*), and use the same reference in both places.

```
"Cathay".substring(3) == "hay"
"Cathay".substring(3).equals("hay") // true
```

Morever: string-literals with + are computed at compile-time.

(*) This optimization is only safe because Java strings are immutable.

trivia: Interning Java string-literals

Literals occur in the source-code text, and can be processed at compile-time. In Java, string literals are "interned": If the same string-literal occurs twice, the the compiler is smart enough to only make one object(*), and use the same reference in both places.

```
"Cathay".substring(3) == "hay"       // false
"Cathay".substring(3).equals("hay") // true
```

Morever: string-literals with + are computed at compile-time.

(*) This optimization is only safe because Java strings are immutable.

trivia: Interning Java string-literals

Literals occur in the source-code text, and can be processed at compile-time. In Java, string literals are "interned": If the same string-literal occurs twice, the the compiler is smart enough to only make one object(*), and use the same reference in both places.

```
"Cathay" == "Cathay"
"Cathay".substring(3) == "hay"      // false
"Cathay".substring(3).equals("hay") // true
```

Morever: string-literals with + are computed at compile-time.

(*) This optimization is only safe because Java strings are immutable.

trivia: Interning Java string-literals

Literals occur in the source-code text, and can be processed at compile-time. In Java, string literals are "interned": If the same string-literal occurs twice, the the compiler is smart enough to only make one object(*), and use the same reference in both places.

```
"Cathay" == "Cathay"                      // true (!)
"Cathay".substring(3) == "hay"            // false
"Cathay".substring(3).equals("hay") // true
```

Morever: string-literals with + are computed at compile-time.

(*) This optimization is only safe because Java strings are immutable.

trivia: Interning Java string-literals

Literals occur in the source-code text, and can be processed at compile-time. In Java, string literals are "interned": If the same string-literal occurs twice, the the compiler is smart enough to only make one object(*), and use the same reference in both places.

```
"Cathay" == "Cathay"                          // true (!)
"Cathay".substring(3) == "hay"                // false
"Cathay".substring(3).equals("hay")           // true
```

Morever: string-literals with + are computed at compile-time.

```
"Cat".concat("hay") == "Cathay"
```

(*) This optimization is only safe because Java strings are immutable.

trivia: Interning Java string-literals

Literals occur in the source-code text, and can be processed at compile-time. In Java, string literals are "interned": If the same string-literal occurs twice, the the compiler is smart enough to only make one object(*), and use the same reference in both places.

```
"Cathay" == "Cathay"                          // true (!)
"Cathay".substring(3) == "hay"                // false
"Cathay".substring(3).equals("hay")           // true
```

Morever: string-literals with + are computed at compile-time.

```
"Cat".concat("hay") == "Cathay" // false
```

(*) This optimization is only safe because Java strings are immutable.

trivia: Interning Java string-literals

Literals occur in the source-code text, and can be processed at compile-time. In Java, string literals are "interned": If the same string-literal occurs twice, the the compiler is smart enough to only make one object(*), and use the same reference in both places.

```
"Cathay" == "Cathay"                      // true (!)
"Cathay".substring(3) == "hay"            // false
"Cathay".substring(3).equals("hay")       // true
```

Morever: string-literals with + are computed at compile-time.

```
"Cat".concat("hay") == "Cathay" // false
"Cat" + "hay" == "Cathay"
```

(*) This optimization is only safe because Java strings are immutable.

trivia: Interning Java string-literals

Literals occur in the source-code text, and can be processed at compile-time. In Java, string literals are “interned”: If the same string-literal occurs twice, the the compiler is smart enough to only make one object(*), and use the same reference in both places.

```
"Cathay" == "Cathay"                     // true (!)
"Cathay".substring(3) == "hay"           // false
"Cathay".substring(3).equals("hay")  // true
```

Morever: string-literals with + are computed at compile-time.

```
"Cat".concat("hay") == "Cathay" // false
"Cat" + "hay" == "Cathay"       // true (!)
```

(*) This optimization is only safe because Java strings are immutable.

typing: when?

statically-typed: At compile-time, the types of all names are known.

Usually provided by programmer and checked by type-system; sometimes inferred by the language (ML, Haskell). (Rust, Java, C# also do *some* type-inference.)

dynamically-typed: Only at run-time, the type of every *value* is known.

BUT, a variable might hold values of different types, over its lifetime. Python, javascript, racket, php. To do this, each value (incl. primitive types) includes some extra "tag" bits to indicate its type.

Since hardware really likes IEEE-doubles with all 64 bits, how to keep tag-bits around? Presumably, use a boxed double (`Double`), with enough extra bits in the box to hold tag bits. Better than having "sub-industry-standard" doubles where tag-bits prevent representing the full IEEE range.

static vs dynamic trade-offs

```
int foo() {if (true) return 17; else return "nope";}
```

will never ever lead to a type-error, yet Java's type-system will still reject it.

```
str += (charAt(0)=='\n' ? "<br/>" : charAt(0));
```

is sensible, but Java's type-system will complain: What is the *type* returned by the conditional-expression? Sometimes `String` but sometimes `char`, so type-system rejects – even though **+=** sensible either way (overloaded).

We say the type-system is "Sound", but not "Complete".

Incomplete Type Inference

In Java18, they broke backwards compatability! (kinda)

`List<Character> myLetters = …;`

`myLetters.map(Character::toString)` compiled in Java 17, calling `Character::toString(char c)` (and auto-unboxing).

In Java 18, they overloaded: `Character::toString(int codepoint)`. The type-system suddenly isn't strong enough; it no longer compiles!

Sol'n: figure how to re-write, to appease compiler.

(Maybe not technically breaking backwards-compat of language-spec, but the provided-compiler-implemention didn't keep up.)

typing: other approaches

duck typing: Care about an object having a field/method, not any inheritance.

E.g. javascript

untyped:

E.g. assembly

type-safe: Any type error is caught (either dynamically or statically).

Note that C is not type-safe, due to casting. Java's casting *is* type-safe(*) — a bad cast will fail at run-time.

(*) Actually, Java generics + casting *can* bypass type-safety, due to type-erasure. :-(

typing: strong/weak/non

These terms are often used in different ways:

strongly typed: no/few implicit type conversions,
or statically typed

weakly typed / untyped: many implicit type conversions,
or dynamically typed

Consider Java **`Math.sqrt(16)`**, or

`"we have " + n+1 + " cookies"` (what if "`n-1`"?)

Cf. SQL (each column strongly-typed) vs SQLite (may attempt type-conversion, but will allow storing any type in a column).

Implicit conversions are often one way "scripting" languages are more lightweight.

Compiling

- A *compiler* is a function

  compile : source-code → machine-code

  The resulting machine-code, when executed, runs the program which produces a resulting value.

"Correctness": the result-code has identical semantics to source-code.

Compiling

- A *compiler* is a function

  compile : source-code → machine-code

  The resulting machine-code, when executed, runs the program which produces a resulting value.

- A *cross-compiler* is just *source-code → machine-code* where the machine-code produced be for a different platform than the one the compiler is running on. (A boring and archaic distinction.)

"Correctness": the result-code has identical semantics to source-code.

Compiling

- A *compiler* is a function

  compile : source-code → machine-code

  The resulting machine-code, when executed, runs the program which produces a resulting value.

- A *cross-compiler* is just *source-code → machine-code* where the machine-code produced be for a different platform than the one the compiler is running on. (A boring and archaic distinction.)

- A *transcompiler* is *source-code → source-code*, so "compile Rust into javascript" is sensible. Machine code is just one example of an target-language, so this subsumes both previous terms.

"Correctness": the result-code has identical semantics to source-code.

Compiling vs Interpreting

- $compile : source\text{-}code \rightarrow source\text{-}code$

  Btw, this general formulation is what people typically mean by "compilation".

Compiling vs Interpreting

- *compile : source-code → source-code*

  Btw, this general formulation is what people typically mean by "compilation".

- An *interpreter* is a function

  eval : expr → value

  which evaluates an expression, producing a result.

Compiling vs Interpreting

- *compile : source-code → source-code*

  Btw, this general formulation is what people typically mean by "compilation".

- An *interpreter* is a function

  eval : expr → value

  which evaluates an expression, producing a result.

  - Interpreted code: CPU runs the interpreter's op-codes; they look at the source-expression *as data*, updating internal state appropriately.

Compiling vs Interpreting

- *compile : source-code → source-code*

  Btw, this general formulation is what people typically mean by "compilation".

- An *interpreter* is a function

  eval : expr → value

  which evaluates an expression, producing a result.

  - Interpreted code: CPU runs the interpreter's op-codes; they look at the source-expression *as data*, updating internal state appropriately.

  - Compiled code: CPU runs the op-codes of the desired program directly.

Compiling vs Interpreting

- *compile : source-code → source-code*

  Btw, this general formulation is what people typically mean by "compilation".

- An *interpreter* is a function

  eval : expr → value

  which evaluates an expression, producing a result.

  - Interpreted code: CPU runs the interpreter's op-codes; they look at the source-expression *as data*, updating internal state appropriately.

  - Compiled code: CPU runs the op-codes of the desired program directly.

  - Compiled code: faster, but platform-specific.

Compiling vs Interpreting (cont.)

The distinction is practical, but not fundamental. You can even view CPUs as interpreters for for compiled-code (!) — they look at the op-codes as data, updating the CPU's state appropriately.

- A compromise: compile to *byte code*; then interpret that byte code. Trades off speed *vs.* platform-dependence. (See also: *JIT*.)